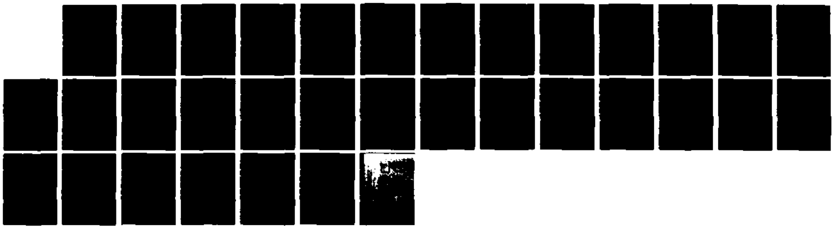
AD-A133 155 EXPERT COMPUTER SYSTEMS FOR MISSILE MAINTENANCE(U) 1/1
MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE
L DAVIS ET AL. 09 AUG 83 N60921-82-C-0083

UNCLASSIFIED F/G 9/2 NL



1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

AD-A133 155

FINAL REPORT: EXPERT COMPUTER SYSTEMS FOR MISSILE MAINTENANCE

Principal Investigators:

Larry Davis
Nick Roussopoulos
Dana Nau
Raymond Yeh

Graduate Research Assistants:

Emily Kasif
Olaf Schoenrich

Computer Science Department
University of Maryland
College Park, MD 20742

Final report for work accomplished on
Naval Surface Weapons Center contract N60921-82-C-0083

83 09 28 036

Table of Contents

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A133 155	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Expert Computer Systems for Missile Maintenance		5. TYPE OF REPORT & PERIOD COVERED Final Aug 82 - Aug 83
		6. PERFORMING ORG. REPORT NUMBER -
7. AUTHOR(s) Larry Davis, Nick Roussopoulos, Dana Nau, Ray Yeh, Emily Kasif, Olaf Schoenrich		8. CONTRACT OR GRANT NUMBER(s) N60921-82-C-0083
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Dept. Univ. of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Surface Weapons Center Attn: R42 White Oak, Silver Spring, MD 20910		12. REPORT DATE 9 Aug 1983
		13. NUMBER OF PAGES 26
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence, missile maintenance

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

New paradigms for failure hypothesis formation are considered. An automated maintenance manual is developed and discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. INTRODUCTION

This is a final report for work accomplished on Naval Surface Weapons Center contract N60921-82-C-0083, "Expert Computer Systems for Missile Maintenance". We have done work on several related topics. Section 2 describes work which was done jointly with McDonnell Douglas, Section 3 describes an Automated Maintenance Manual which was developed under this contract, and Section 4 describes results we have obtained on a number of longer-range issues related to missile maintenance.

2. JOINT WORK WITH MCDONNELL DOUGLAS

Since this contract was awarded to the University of Maryland in conjunction with a similar award to McDonnell Douglas, some of the work on the contract was done jointly with McDonnell Douglas. This section summarizes the nature of the work that was done jointly.

2.1. Acquainting McDonnell Douglas With Expert Systems

In order to familiarize McDonnell Douglas with the general techniques used in expert computer systems, and also to keep them informed of the work we were doing, we provided them with the following papers:

1. Nau, "Expert Computer Systems" [8]. This paper is a general survey of the techniques most commonly used in expert systems.

2. Nau, Reggia, and Wang, "Knowledge-Based Problem Solving Without Production Rules" [10]; Reggia, Nau, and Wang, "Diagnostic Expert Systems Based on a Set Covering Model" [14]; and various related internally generated memos. These papers and memos describe an approach to diagnostic problem solving based on a set covering model. Developed originally by James Reggia and currently being investigated by Reggia and Nau (one of the principal investigators on this contract), this approach has been used successfully in medical diagnosis systems, and in the general purpose expert system KMS.

3. Allen, "Yaps: Yet Another Production System" [1]; Wood, "Franz Flavors: An Implementation of Abstract Data Types in an Applicative Language" [15]; Allen, Trigg, and Wood, "The Maryland Artificial Intelligence Group Franz Lisp Environment" [2]. (We also provided McDonnell Douglas with a computer tape of the three systems described in these three reports.) Yaps is an antecedent driven production system based on discrimination nets. Yaps is similar to, but more general than, OPS5 [4]. Franz Flavors is an implementation in Franz Lisp of the Lisp-Machine Lisp Flavors package developed at MIT. Flavors are an ideal medium for the generation of object oriented systems such as constraint networks. The Franz Lisp Environment is a 'front end' developed at Maryland for Franz Lisp. It provides useful macros and a history mechanism to facilitate working in Lisp.

2.2. Investigation of a Particular Application

At the suggestion of McDonnell Douglas, the Harpoon missile guidance system power supply was chosen as an example circuit for use in testing the diagnostic techniques being developed. This circuit is built and maintained by IBM at their plant in Owego, New York. In order to familiarize ourselves with the operation of this circuit, and with the diagnostic methods used by the technicians responsible for maintaining it, we decided that we should visit Owego. On October 20, 1982, Dana Nau and Olaf Schoenrich from the University of Maryland and Jim Miller from McDonnell Douglas spent the day with the Owego technicians. This trip was useful in several respects:

1. Insight was gained into the methods used by the technicians to diagnose a faulty circuit.

2. We learned what kind of automatic testing and diagnostic equipment the is available to the technicians, and

3. The technicians provided information about what kind of automatic testing system they would like to see.

Conversations with technicians reveal that they do electronic testing and diagnosis in a hierarchical way. A circuit board is usually thought of as a collection of functional subunits, each of which is a collection of sub-subunits organized in a certain way. An experienced technician has knowledge not only of individual electronic components, but also of the various subunits, sub-subunits, etc., and the way they interact with each other. Given a circuit board to diagnose, a technician will perform a number of tests to tell whether it is working correctly. If it fails these tests, then one or more of the subunits is faulty. The technician will use the test results, together with his/her knowledge of the subunits and the ways they interact, to select which subunits to consider. The same diagnosis procedure is then performed on these subunits.

The automatic testing equipment that the technicians have at their disposal does not lend itself to this kind of testing. The fully automatic testing equipment will, when connected to a circuit board, perform an elaborate predefined set of tests. It will make all the necessary measurements, even read oscilloscope outputs, and it will display the results. When it has located an error, the tester will stop and signal what has been found. The chief objection on the part of the technicians is that there is no way to override the testing sequence. Much of the time, the technician will have a good idea of what is wrong, or of what should be tested next, but there is no way to change the sequence of tests performed by the automatic tester.

What the technicians want, it would seem, is a system that will aid them in testing, but where they are still 'in control'. That is, they would like a system that will make it easy for them to use a hierarchical testing strategy.

After the trip to Owego, the technicians at McDonnell Douglas studied the power supply circuit and produced a preliminary set of production-like rules describing the correct behavior of the circuit, as well as its behavior given certain errors. These preliminary rules were quite useful to us, but they suffered from two major drawbacks. They were not complete in that they only described the manifestations of certain kinds of errors. They also did not show how a failure in one component would effect the rest of the circuit.

Nonetheless, the rules were important and showed how a technician thinks of errors and their manifestations. This influenced the development of the paper "Knowledge-Based Problem Solving Without Production Rules" [10], which was produced under this contract. A copy of that paper is included as an appendix to this report.

3. AUTOMATED MAINTENANCE MANUAL

The task of building a completely automatic computerized diagnostic system is beyond the current state of the art. Therefore, means must be provided for the machine maintenance personnel to interactively involved in diagnosis process. As an intermediate solution for the fault diagnosis problem, we envision a maintenance computer system that will allow the engineer to specify a functional description of the object to be diagnosed and from that specification the system will automatically create a set of tests that will assist maintenance personnel in their troubleshooting. It is likely that some of these tests may be performed by the maintenance system itself.

To achieve this goal, several problems must be addressed and solved. The first problem is designing a mapping between the functional description of the device to be maintained and a set of tests that would locate the possible failures once a problem was encountered. The second problem is a design of a coherent language to represent the set of tests to be performed during the troubleshooting procedure. The third problem is inherently a control problem since it involves dynamic execution of the proper tests depending on the knowledge that was accumulated by the computer system as a result of performing previous tests.

3.1. AMM System Overview

We have have designed a computer system, an Automated Maintenance Manual (AMM), to provide a flexible tool for specifying a series of tasks to be performed by the servicing personnel in order to diagnose faults of a device. The system can be used as a tool for identifying bugs in the device as well as for periodic preventive maintenance procedures. The AMM system is interactive. It allows the user to record the results of his inquiries and provides the user with guidance as to where the potential problem may be. With a few simple commands, the engineer can now specify an entire series of diagnostic procedures in a particular format which should help technicians to isolate where the device failure lies (see Figure 3.1).

An AMM system is comprised of a knowledge base and a control structure. The knowledge base is organized hierarchically and contains tests, text, and diagrams (see Figure 3.2). Tests are diagnostic procedures which constrain the set of possible faults that could exist in a device. Text is free-form data which contains general information appropriate to a particular test. Diagrams are visual representations which correspond to the test being administered. All of this information will be available to the technician during a troubleshooting session. The control structure of the system provides the technician with guidance during the diagnostic session directing him to the next appropriate set of tests to be administered. It also contains the provision for the technician to take the initiative and guide a troubleshooting session himself.

The AMM system is, therefore, an intermediate solution to the maintenance problem where a complete solution would involve a system which could automatically create a partially ordered set of diagnostic tests, given a specification of the device and a problem to be diagnosed. The system we have designed and implemented is a consultation program which provides a link between the device manual specification and the set of tests used to analyze and identify problems. These tests are related to the inherent structure of the device being diagnosed.

Maintenance manuals are often written in a format useful for locating specific information about a particular device. AMM provides a uniform language which links diagnosis of a device's problems and the device's manual. This command language is flexible enough to be used for a large class of servicing procedures and may be used naturally by the engineer who designates diagnostic procedures.

Using command language, the engineer can create the basic components of an AMM system: decision nodes, control arcs, packages, and a dictionary. Decision nodes are tests used to limit the set of possible faults which could have caused a device to fail. Control arcs are directed edges leading out of a decision node and direct system control to the next appropriate decision node or package. Packages are composed of a set of decision nodes connected by control arcs. All nodes within the same package are associated with isolating one specific problem in the hierarchy of problems that can cause a device to fail. An AMM dictionary contains a list of all packages and decision nodes contained at each level in an AMM hierarchy. In particular, this dictionary contains the name of each element, its address in memory, the number of elements, the root of this particular sublevel, and the alphabetical order of the element list.

3.2. AMM Organization

The AMM knowledge base is organized hierarchically into several major components: decision nodes, control arcs and packages, along with a dictionary.

3.2.1. Decision Nodes

The basic component of AMM is the decision node (refer to Figure 3.3). A decision node must characterize the ideal function of a particular state in the system. The decision function f is a partial function of discrete variables $x_1, \ldots, x_n$, where each variable x_i takes exactly m_i values. In the case where f is a constant, the decision tree for f is composed of a single decision node labeled by a constant value. In practicality this would mean that no matter what the outcome of a diagnostic procedure, the system would be directed towards only one result. If f is null, then the decision tree for f is a single leaf labelled with the null symbol. In terms of AMM, a null function f would mean that the program had been compiled with nothing in the input file and thus there would be no decision tree made. In all other cases, for each x_i, $1 \leq i \leq n$, such that at least two restrictions (say, $f \mid x_i = k_1$ and $f \mid x_i = k_2$) are not null, f has one or more decision trees composed of a root labeled x_i and m_i subtrees.

At every decision node there are two possible conditions which may exist. In the first instance, we know the source of the devices failure. If this is the case then at the decision node we must output the solution or more formally, the function value. In the second case, we do not know what the source of the device failure is. We make a prediction and design a sequence of steps to narrow the number of possibilities which might cause manifestations of device problems. A prediction may be considered to be a disjunction of possible device failures.

3.2.2. Control Arcs

A control arc is the directed edge leading out of a decision node. There is a one to one correspondence between the number of control arcs leaving each decision node and the number of exits from each node. Each control arc is associated with a semantic action. The actions direct system control to the next appropriate decision node/package and emit portions of the text associated with each arc.

3.2.3. Packages

A package is a fundamental building block in the AMM system used to achieve modularity and flexibility in the system organization (refer to Figure 3.4). Packages may correspond to a set of troubleshooting procedures for testing a module in the device being diagnosed. Packages are composed of a set of decision nodes connected by control arcs. All nodes in a package have the same generic package name as well as a specific decision node name. All nodes within the same package are associated with isolating one specific problem in the hierarchy of problems that can cause a device to fail. There may be multiple occurrences of the same functional package in different parts of the decision tree, but every package within the system must have a unique name to identify its individual role in the troubleshooting process.

In purpose, a package may be considered analogous to a procedure in a computer program. Like a procedure, a package should contain only a limited number of procedure steps or in the case of a package, decision node. Its function is to satisfy one specific task in the schema of jobs that must be performed in the entire unit. It is the encapsulation of one particular module within the network of modules in the system.

One may also think of a package as a higher level decision node. A decision node has one entry port and possibly multiple exit ports to other decision nodes,and a package also has only one entry port and may have multiple exits. Like a decision node, a package represents the partial function of mapping variable x_i to m_i possible outcomes. This function f may be constant or null in which case the package represents a constant value or is empty. Or in the more general case, it represents at least two or more possible restrictions corresponding to m_i outcomes. As in the case of the decision node, at every package we have two possible conditions which may exist. In the first instance, we know the source of the device's failure in which case the package outputs the solution or conclusion reached during the diagnosis process. In the second case, the source the devices problems is not known so control is transferred to another package to continue troubleshooting.

3.2.4. AMM System

An AMM System consists of a set of packages which are connected by control arcs (refer to Figure 3.5). As in the case of decision nodes, and packages, an AMM system contains one entry port and multiple exit ports. Like a decision node, and packages, an AMM system represents the partial function of mapping variable x_i to m_i possible outcomes. This function f may be constant or null in which case AMM represents a constant value or is empty. Or in the more general case, it represents at least two or more possible restrictions corresponding to m_i outcomes.

3.2.5. Dictionary

Corresponding to every level in the AMM hierarchy is a list of all elements to be found at that level (refer to Figure 3.6). At the top level is the AMM dictionary which specifies the root of the system. In particular, this dictionary contains the label of each element listed at that sublevel, its address in memory, the number of elements currently listed at this particular sublevel of the hierarchy, the root of this particular sublevel, and the alphabetical order of the element list. Whenever an element from this level is required, it is searched for using the binary search technique. Since the elements are already ordered alphabetically, they may easily be listed for the technicians benefit in alphabetical order without requiring a new sorting of elements which would have been the case had, for instance, a hash function been used to sort elements in a list.

The root of the hierarchy at the top level points to the next level in the hierarchy which contains the list of packages currently in the system. The dictionary at the package level contains a list of every package label in the system in the order that the packages were created, the number of packages in the system, the root decision node of that package, the address of the root decision node in memory and a sorting of package labels in alphabetical order for easy access.

Every package listed in the dictionary also has a pointer to the list of all decision nodes within that package. This list of a package's decision nodes is the next lower level in the dictionary. Decision node dictionaries contain along with the list of decision node label in order of creation, the number of decision nodes in that package, their location in memory, and the alphabetical order of the decision nodes belonging to that one package.

So from the above description we can deduce that there exists only one dictionary at the package level of the hierarchy whereas there exists n dictionaries at the decision node level, one decision node dictionary for every package in the system. Every element in a dictionary must have a unique name in order for the binary search to be successful. Thus all packages must have unique names and all decision nodes within a particular package must have a unique name. However, decision nodes in different packages in the system may have identical names. The possibility for identically named decision nodes in the system exists because the binary search is performed only upon the the decision nodes within one package.

This provision was created to allow engineers to create multiple occurrences of the same package in different parts of the decision structure without having to rename the decision nodes within the same functional package. Only the different occurrences of the package must have unique names. One may compare this organization to the labelling of variables in computer programs where local variables in a procedure must have unique names but there may exist this identical local variable name in another procedure in a program. Every procedure in a program must have a unique name just as every package in AMM must be uniquely labelled.

3.3. AMM Conclusions

The Automated Maintenance Manual is designed to assist engineers in documenting diagnostic procedures for maintaining systems and to assist technicians in diagnosing device failures. AMM is organized in a decision tree format which corresponds to many of the documents already in existence. Decision trees model a discrete function where the value of the current variable determines the next variable to be evaluated or outputs the function's value.

AMM was designed to help alleviate some of the drudgery of creating documentation. By mastering a few commands in the command language syntax the engineer can now specify a series of diagnostic procedures. He has great flexibility in formulating the actual tests to be performed, specifying the order in which tests a[illegible]e conducted,listing the correct range of proper respons[illegible] to a [illegible]articular experiment, eliminating unnecessary tests during a [illegible] testing sequence, and in limiting the domain which his tests a[illegible] supposed to diagnose.

Further information about AMM can be found in [5], a copy of which is available on request.

4. LONG-RANGE ISSUES FOR DIAGNOSIS AND MAINTENANCE

In Section 3 we described a system implemented under this contract which automates a portion of the diagnosis and maintenance process, and which makes use of information already gathered by McDonnell Douglas under this contract. However, it is clear that much more can be done in the use of AI techniques for automation of diagnostic and maintenance tasks. As described in this section, we have been investigating several important theoretical issues concerning the applicability of various AI techniques to electronic diagnosis. Section 4.1 contains some introductory comments about the nature of electronic diagnosis, and Sections 4.2 and 4.3 describe our investigations of two approaches to the diagnostic problem.

In addition, it should be noted that missile maintanence is intimately linked with missile manufacturing. Thus we have directed some of our research to the use of artificial intelligence in automated manufacturing. This work is described in detail in the paper "Prospects for Process Selection Using Artificial Intelligence" [9] which was supported by this contract and is included as an appendix to this report.

4.1. The Hierarchical Nature of Electronic Diagnosis

Conversations with technicians reveal that they do electronic testing and diagnosis in a hierarchical way. A circuit board is usually thought of as a collection of functional subunits, each of which is a collection of sub-subunits organized in a certain way. An experienced technician has knowledge not only of individual electronic components, but also of the various subunits, sub-subunits, etc., and the way they interact with each other. Given a circuit board to diagnose, a technician will perform a number of tests to tell whether it is working correctly. If it fails these tests, then one or more of the subunits is faulty. The technician will use the test results, together with his/her knowledge of the subunits and the ways they interact, to select which subunits to consider. The same diagnosis procedure is then performed on these subunits.

The procedure described above suggests an approach for use in knowledge-based electronic diagnosis. One could imagine a knowledge-based diagnostic procedure operating roughly as follows:

```
proc DIAGNOSE(M):  /* M is an electronic module */

    Use knowledge about M to select various diagnostic tests.

    Perform these tests on M.

    If M performs correctly, then return.

    If M performs incorrectly and M has no functional subunits, then
```

```
        M is faulty.  Replace M and return.

        Use knowledge about M to construct the set S = {M_1,M_2,...,M_n} of
        the major functional subunits of M.

        Using the test results, and knowledge about the members of S,
        remove from S some of its members (those known to be OK).  Let
        S′ be the set of all remaining members.  Order the members of S′
        in terms of which of them is most likely to be faulty.

        For each M′ in S′, call DIAGNOSE(M).

    end DIAGNOSE
```

The procedural framework described above omits many important details. For example, such a system would require considerable knowledge about the nature of each of the functional subunits, sub-subunits, etc. Just how this knowledge would best be used and how it would best be organized is a very important question. We have been examining two possible approaches, as described in Sections 4.2 and 4.3.

4.2. Propagation of Constraints

Propagation of constraints is one of the main techniques used in most AI electronic diagnosis systems. It is an extremely powerful modeling formalism and can be used not only to determine the outputs of a circuit from the inputs, but also to determine the inputs from the outputs, as well as intermediate circuit values. It also frequently happens that, using a constraints network, fewer values are needed to make these determinations than would be needed in an actual circuit.

A constraint network can be defined mathematically as a directed graph $G = (V,E)$ for which a variable is associated with each vertex in V, and a predicate (called a constraint) is associated with each edge in E. Given some initial values for some of the variables, values (or sets of possible values) for the other variables are determined by the requirement that all of the predicates must be satisfied simultaneously.

This mathematical abstraction can be used in modeling many kinds of systems. For example, if an electronic component is associated with each vertex in the network, and each edge is thought of as wire between components, then a constraint network is an excellent model for a circuit. As a simple example, consider the adder shown below:

```
(4.2.1)          x---| ‾‾‾‾‾ |
                     |adder|---z
                 y---|_____|
```

The constraints (predicates) that would be associated with this adder are:

```
(4.2.2)          z = x + y
                 y = z - x
                 x = z - y
```

Thus, if values were given for any two of x, y and z, the constraint network would automatically generate the third value using one of the rules listed in (4.2.2).

Rules, or rather, constraints, of this type can be written for other components. Consider a multiplier:

```
(4.2.3)          x---|‾‾‾‾|
                     |mult|---z
                 y---|____|
```

In this case, the constraints are:

```
(4.2.4)          z = x * y
                 y = z / x
                 x = z / y
```

The above examples are, of course, trivial, and do not demonstrate the power of a constraint network. It is only when several components are connected that this power becomes apparent. Consider the circuit show below:

```
                               5
(4.2.5)          x---|‾‾‾‾‾|   |
                     |adder|--*--|‾‾‾‾|
                 y-*-|_____|     |mult|------z
                   |_____________|    |
                                 |____|
```

This circuit expresses the relationship:

```
(4.2.6)          x + y = 5
                 y * 5 = z
```

There are two things to note in this example. First, note that if any one of x, y or z is provided, the constraint system can determine the other two values. For example, if x is know the system uses the rule from (4.2.2)

```
                 y = z - x
```

to determine the value of y. It then uses the rule from (4.2.4)

```
                 z = x * y
```

to determine z.

The second thing to note is that while the circuit is making the proper determinations for x, y and z, it is not necessarily doing this the same way the actual circuit would. This is because, in a constraint network, information is free to travel in both directions; from inputs to outputs, and from outputs to inputs. While the rule 'y = z - x' expresses a true fact about adders, no real adder can perform subtraction. A constraint network, however, is free to use this rule to actually compute what y must be in order to get z and x.

This computation of values would be of little use if the values could not be changed and the results examined. The question naturally arises, then, of what happens if, say, one of the inputs changes. As an example, consider the following circuit for converting from centigrade to Fahrenheit:

```
                        ____   a    ____    b         _____
(4.2.7)         C---|    |-----|    |-----------|     |
                    |mult|     |mult|           |adder|-----F
                9---|____|  5--|____|     32---|_____|
```

where C is the centigrade temperature, F is the Fahrenheit temperature, 9, 5 and 32 are all constants, and a and b are labels.

If a value of 100 is given to C, then a gets a value of 900, b gets a value of 180 and F gets a value of 212 degrees. Suppose now that we want to give C a value of 0. By the first rule in (4.2.4) we should have x * y = z, but in this case that would give us 0 * 9 = 900, which is wrong. In general, to avoid this problem, constraint systems would retract all the values that were derived from the original value for C. Thus, a, b and F would again have no value and there would be no problem with asserting a value of 0 for C.

Simply retracting values is no longer adequate when there is looping or when there are multiple sources for a value. Such a situation occurs in an SR flip-flop.

```
                     ___
                S---|   |
(4.2.8)            _|nor|_
                  | |___| |
                  |       |b
                  |       |___ ___
                a |           |nor|---*--Q
                  |       R---|___|   |
                  |___________________|
```

When values of, say, 1 and 0 are given to S and R we have the following: the value of 1 for S constrains b to be 0. This 0, with the 0 on R, constrains Q to be 1, which also makes a 1. What happens in a real flip-flop at this point, if S is set to 0, is that the 0 on S and the 1 on a do not change the value of 0 on b, so the output Q stays 1. In the constraint network, if we follow the strategy of the temperature converter and remove the values from b and a, when S is set to 0 there is no way to tell if b should be 0 or 1, as there is no value on a.

Clearly there is a problem. In circuits with loops, such as the flip-flop, it is inadequate to simply retract intermediate values. It is unfortunately not clear what should be done instead. One possibility is to retract a value only when a new one is being propagated over an old one. This would handle the flip-flop case, but it is not clear whether it will work in general. Another possibility is to include time in some fashion. This is something that is desirable for other reasons as well.

To see why it is desirable to include time, suppose we now look at circuits containing time-dependent devices such as memory units or delay lines. Then an ordinary constraint network such as those discussed above will not suffice. This is because constraint networks (as we have defined them above) can only model situations where every constraint must be satisfied simultaneously, whereas memory units and delay lines have constraints on future or past values in terms of present ones. For example, consider the following oscillator.

```
                     |\
                     | \
          ___________|  >O_____
(4.2.9)  |     a     | /   b   |
         |           |/        |
         |                     |
         |      _________      |
         |     |         |     |
         |_____|  delay  |_____|
               |_________|
```

The constraint on the NOT gate is "a = ~b", and the constraint on the delay unit is "the next value of a will be the current value of b".

Here there is a problem similar to the problem with the flip-flop; if all values are retracted then there is no way to determine what the new value will be. If values were retracted properly, it should be possible to have the constraint network cycle in a way that simulated the behavior of the real oscillator being modeled.

Another possibility, however, is to include time specifically as a variable in the constraints; in this case the constraint through the delay would be:

(4.2.10)
$$(a, t) = (b, t-1)$$
$$(b, t) = (a, t+1)$$

(This should be read "a at time t equals b at time t-1".)

This would create an infinite network rather than a finite one. While this might at first seem an unwieldly concept, it has the advantage that it appears to simplify somewhat the problem of determining just what the circuit does.

In some cases the problem of recognition is straightforward, or at worst requires only some algebraic manipulation. For example, in

(4.2.5) only simple algebra is needed to determine that the global constraint is

$$xy + y^2 = w,$$

and it is also simple to determine from the temperature converter that

$$C = (5/9)(F - 32).$$

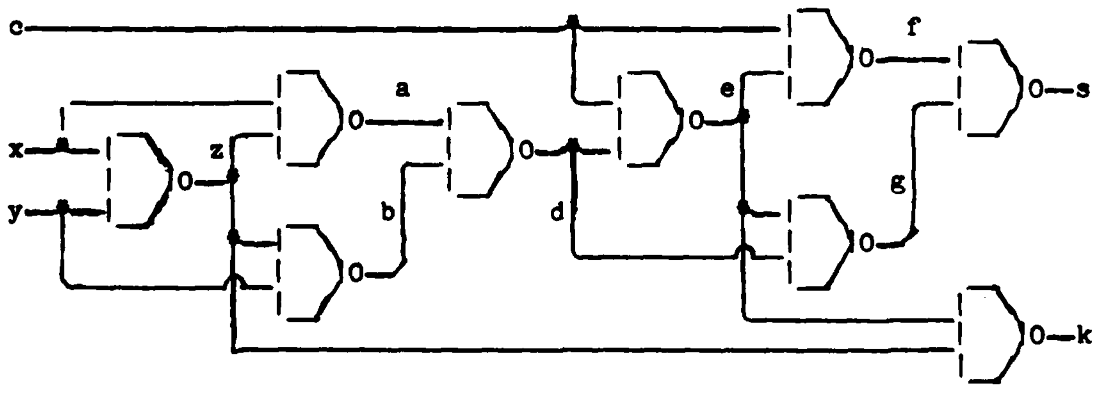
It is not so simple in cases where the behavior is time dependent. In the case of the oscillator (4.2.9), realizing that it oscillates requires noticing that the entire state of the system at time t+2 is the same as it was at time t, whence the state at times t+2, t+4, ..., must also be the same. This looks doable in some sense, although it is not clear how it would be done if the oscillator were part of a larger system.

A similar sort of thing would have to be done in the case of the flip-flop (4.2.8). Even though this does not oscillate, it is still time dependent in that it remembers which of S and R was last high. In this, and other devices with memory such as counters, it seems critical to include memory in the constraints.

Constraints in a constraint network are usually implemented as pattern-invoked programs. The invocation of these programs is data-driven; i.e., a constraint P(x1,...,xn) is only invoked if enough of the variables x1,...,xn are known that all of the others can be determined by applying P.

As an example, consider the binary adder given below, which is made out of 9 NAND gates. x, y, and c are the inputs (c is the carry input), s is the output, and k is the carry output.

(4.2.11)



The constraints describing this network are the following:

```
(4.2.12)        z=x@y;  a=x@z;  b=y@z;  d=a@b;  e=c@d;  f=c@e;  g=d@e;

                s=f@g;  k=z@e;  c,x,y are in {0,1};
```

where "@" is the NAND operation.

At first glance, prolog seems like a natural vehicle in which to implement such constraints. The constraints defining a NAND gate can be described in prolog as

```
    (4.2.13)        nand(0,0,1).
                    nand(0,1,1).
                    nand(1,0,1).
                    nand(1,1,0).
```

The description of the adder would thus be

```
(4.2.14)     adder(x,y,c,s,k) <- nand(x,y,z) & nand(x,z,a) &
                nand(y,z,b) & nand(a,b,d) & nand(c,d,e) &
                nand(c,e,f) & nand(d,e,g) & nand(f,g,s) & nand(z,e,k).
```

Thus, under the initial conditions x=1, y=0, c=1, the problem of finding the values of s and t in the adder could be coded in prolog as

```
(4.2.15)        <- adder(1,0,1,s,k) & write(s) & write(k).
```

However, the operation of prolog is a bit different from that of most implementations of constraint networks. As mentioned earlier, constraint networks normally are data driven: a constraint P(x1,...,xn) is only applied if enough of the variables x1,...,xn are known that the others can be determined by applying P. Thus if it were known that x=1,y=0, and c=1 in the adder, the constraints would be applied in the order

```
(4.2.16)        first   nand(x,y,z);
                second  nand(x,z,a), nand(y,z,b);
                third   nand(a,b,d);
                fourth  nand(c,d,e);
                fifth   nand(c,e,f), nand(d,e,g), nand(z,e,k);
                sixth   nand(f,g,s).
```

However, prolog does a depth-first search going top down from the predicates in the Horn clauses, regardless of where the data is. For the adder definition given in (4.2.14), prolog would behave in almost the same way as a data-driven search; the order of instantiation would be similar to the order given in (4.2.16). However, if the definition of the adder given in (4.2.14) were replaced by

```
(4.2.17)    adder(x,y,c,s,k) <- nand(f,g,s) & nand(d,e,g) &
                nand(c,e,f) & nand(c,d,e) & nand(a,b,d) &
                nand(y,z,b) & nand(x,z,a) & nand(x,y,z) & nand(z,e,k).
```

then prolog would behave in almost the same way as a goal-directed search. This has both advantages and disadvantages.

The main advantage of prolog's search is that it is more general than a data-directed search: prolog may be able to deduce an answer where an ordinary constraint propagation system cannot. For example, if s=1 and k=0, then prolog can determine that one of the following must hold:

```
(4.2.18)        x=0  y=0  c=1  s=1  k=0
                x=0  y=1  c=0  s=1  k=0
                x=1  y=0  c=0  s=1  k=0
```

There are two main disadvantages to prolog's search. First, prolog may require considerably more computation than constraint propagation. Clearly, a depth-first search can be very wasteful at times, and examples have been given in the literature to show that prolog can sometimes do much unneeded searching. Second, if prolog's built-in arithmetic predicates are used, an error occurs if they are invoked with more than one variable uninstantiated. For example, consider the data-flow network given below (x, y, z, and w are arbitrary integers).

```
                                 z
                    ______
(4.2.19)        x---|      |     |
                    |adder |--*--|‾‾‾‾‾‾|
                y-*-|______|     |adder |------w
                  |______________|      |
                                 |______|
```

This can be defined in prolog as

```
(4.2.20)        network(x,y,z,w) <- plus(y,z,w) & plus(x,y,z).
```

where plus(p,q,r) is a built-in predicate that is true if and only if p+q=r. If it is known that z=2 and w=3, then both prolog and constraint propagation can determine that x=y=1. However, suppose we redefine the network as

```
(4.2.21)        network(x,y,z,w) <- plus(x,y,z) & plus(y,z,w).
```

This definition is logically equivalent to (4.2.20) and a propagation of constraints system would behave exactly as before. However, prolog cannot determine that x=y=1 using (4.2.21); instead, an error message results when "plus(x,y,2)" is invoked.

It should be noted that regardless of which of (4.2.20) or (4.2.21) is used, neither constraints nor prolog can determine y and z given x and w. However, suppose that instead of using a built-in predicate "plus" we defined it using Peano's postulates as

```
(4.2.22)        plus(0,p,p).
                plus(S(p),q,S(r)) <- plus(p,q,r).
```

Then prolog could determine that y=S(0) and z=S(S(0)) given x=S(0) and w=S(S(S(0))), whereas ordinary propagation of constraints still could not determine anything.

The optimal approach would appear to be the following: use data-driven propagation of constraints whenever possible. When no more existing constraints can be applied, then start a prolog-style search.

4.3. Set Covering

4.3.1. Deficiencies in Current Rule-Based Diagnostic Systems Below is an excerpt from the paper "Knowledge-Based Problem Solving Without Production Rules" [10] which was supported by this contract and which was presented at the IEEE Trends and Applications conference this May:

> Despite the wide use of rule-based reasoning, it is not clear that production rule systems in general are entirely adequate. Production rules have long been criticized as a general means of knowledge representation ...
>
> In expert computer systems for inferential problems ..., the required format for production rules is usually along the lines of
>
> IF manifestations (or other evidence)
> THEN conclude cause.
>
> However, most of the knowledge which humans use to create such rules goes in the opposite direction: if some cause is present, then certain manifestations will occur. For example, as part of a research project in electronic diagnosis [6], a group of electronic technicians were given a brief introduction to the concept of production rules, including an example of the kind of rules used in Mycin. They were then asked to write down some production rules describing an electronic circuit for use in electronic diagnosis. The rules they produced were almost all of the form
>
> IF cause
> THEN manifestations
>
> Presumably the technicians were specifying their knowledge in an intuitively familiar form, more along the lines of describing each cause rather than providing fixed rules for diagnosis. One naive way to translate such rules into the previous format would be to interchange the "IF" and the "THEN" clauses, but it is not clear how well this would work--if at all.

The format "IF manifestations THEN conclude cause" used in Mycin means that Mycin is set up to solve the converse of the general diagnostic problem: it finds something implied by the given facts rather than something which implies the given set of facts. Most other diagnosis systems we have seen operate this way. It is probably no wonder why the systems were set up this way--it is easier to do rule-based deduction than to solve a set covering problem--but perhaps this rule format is one reason why users of diagnostic production systems have experienced so much difficulty constructing adequate rules for these systems.

KMS.HT [13] [14] and Internist [7] [12] (which are not rule-based) are the only diagnosis systems we know of which are set up to find something which implies the given set of facts. However, it should be possible to do this using rule-based systems as well, as discussed below.

4.3.2. A Possible Approach for Rule-Based Electronic Diagnosis

Let us examine what kinds of rules are needed for electronic diagnosis. The rules in the preliminary list that McDonnell Douglas gave us [6] are of the following forms:

(4.3.1) cause -> manifestation

(4.3.2) cause1 OR cause2 OR ... OR causeN -> manifestation

(4.3.3) cause -> manifestation1 & ... & manifestationN

Each rule given in form (4.3.2) can be simplified to a number of rules of form (4.3.1):

```
cause1 -> manifestation
cause2 -> manifestation
...
causeN -> manifestation,
```

Each rule given in form (4.3.3) can similarly be simplified to

```
cause -> manifestation1
cause -> manifestation2
...
cause -> manifestationN.
```

Suppose for the moment, then, that all of our rules are of form (4.3.1). Suppose we have a malfunctioning circuit which exhibits manifestations m1, m2, ..., mN. For each mi (i=1 to N), we could do backward chaining from mi until we could go no further. This would give us a set causes(mi) of "root causes of mi", any one of which could (through a chain of cause-and-effect) cause mi.

Knowing the possible "root causes" for each mi would define a causal relationship C between the set of all root causes and the set {m1, m2, causes(m1), causes(m2), ..., causes(mN), such a system could find every minimal set S of causes such that if all causes in S occurred then all of m1, m2, ..., mN would occur. Furthermore, if the system were set up correctly, it would not have to be given causes(m1),...,causes(mN) all at once, but could be given them one at a time as each manifestation is discovered.

There are additional complications, however. It seems likely that we would also need rules of the form

(4.3.4) cause1 & cause2 & ... & causeN -> m,

which cannot be simplified. The causal relationship C could not be determined as described above, since it would no longer be true that each individual "root cause" in causes(m) could by itself cause m. There are at least two possible approaches to this:

1. One possibility would be to replace each such rule with a number of rules of the form

```
cause1 -> m1
cause2 -> m2
...
causeN -> mN,
```

with the implicit understanding that m1 = m2 = ... = mN = m. With the rules in this form, the causal relationship C could be constructed and a minimal set of causes could be found. There are at least two problems with this, however:

(a) There is some question whether a minimum cover for some set of manifestations containing {m1,...,mN} would contain a set of causes sufficient to fit the original rule. We think it would, but this issue needs to be investigated further.

(b) If we have

```
a1 & a2 & ... & aK -> b1
b1 & b2 & ... & bN -> c
```

then it will have to be split into

```
a1 -> b11
a2 -> b12
...
aK -> b1K
b11 -> c11
b12 -> c12
...
b1K -> c1K
b2 -> c2
b3 -> c3
...
bN -> cN,
```

and in general, the number of "equal" manifestations could become quite large.

2. If the rules given in form (4.3.4) were used directly, this would set up an AND/OR graph structure in which each manifestation could be caused not by any of a number of root causes, but rather by any of a number of conjuncts of root causes. All such conjuncts could be represented as separate entities to a system such as KMS.HT, which could then find minimal sets of conjuncts. However, just as approach 1 could result in a very large set of "equal" manifestations, approach 2 could result in a very large set of conjuncts of root causes. There may be some way to cut down on the sizes of these sets, but we have not finished investigating this yet.

In addition to handling rules of the form

cause1 & cause2 & ... & causeN -> manifestation,

it will also probably be necessary to handle negations of causes or manifestations (e.g. "the bulb is not lit" as opposed to "the bulb is lit"), and also to simple arithmetic relations (e.g. knowing that voltage>7 implies voltage>5). This means that complicated tests will be necessary for the backward chaining, but this is not too dissimilar from tests that have already been done in some systems.

4.3.3. Problem 1: A Diagnostic Problem

The above paragraphs have discussed the motivation for looking at diagnostic problems involving a number of causal rules of the form

c1 & c2 & ... & cn => m

where each ci is a disorder or "cause" that can cause certain manifestations and m is a manifestion which occurs when all of c1, c2, actually present in some diagnostic problem, and suppose we want to find all minimum sets of "root causes" which (by appropriate rule chaining) are sufficient to cause every manifestation in M. If we add the rule

m1 & m2 & ... & mN => X

to the system, where X is a symbol which does not appear in any other rule, then the problem becomes the problem of finding a minimum set of "root causes" which is sufficient to cause X.

4.3.4. Problem 2: A Grammatical Problem

The above problem can be translated into a grammatical problem as follows. Let G1 be the context-free grammar such that for every one of the causal rules

c1 & c2 & ... & cn => m

given above, G1 contains a production of the form

```
m -> c1 c2 ... cn.
```

The set of nonterminals in G1 is the set of all symbols appearing on the left hand sides of productions. The set of terminals is the set of all symbols which appear on right hand sides of productions but not on left hand sides. The start symbol is X.

Let L1 be the language generated by G1. The problem is to find every string S in L1 such that S has minimum number of distinct symbols of any string in L1. By distinct terminals, we mean that if the terminal T appears twice in S, it is only counted once.

4.3.5. Set Covering

Problem 2, by the way, is a generalization of the set covering problem. To see this, let T={t1,t2,...,tn} be a set, F = {T1,...,Tk} be a family of subsets of T, and G2 be the grammar whose rules are

```
the rule "X -> t1 t2 ... tn"
for every Ti containing t1, a rule "t1 -> Ti"
for every Ti containing t2, a rule "t2 -> Ti"
    ...
for every Ti containing tn, a rule "tn -> Ti"
```

The start symbol for G2 is X, and the terminals are T1, T2, ..., Tk. Let L2 be the language generated by G2. Note that every string in L2 has length n.

Suppose that the string S = "T2 T5 T2 T7" (which contains 3 distinct terminals) is in L2. Then {T2,T5,T7} is a set cover for T. Furthermore, if the number of distinct terminals in S is less than the number of distinct terminals in any other string in L2, then {T2,T5,T7} is a minimal set cover for T.

4.3.6. AI Search Procedures

The correspondence between context-free grammars and AND/OR graphs is well-known. Thus one could consider modifying AI problem-reduction search procedures such as AO* [11] to work on Problem 2. This is discussed below.

If AO* were used unmodified, it would find a string S in L of minimum cost, where the cost was taken to be the sum of arbitrary costs assigned to the productions used to generate S. If the cost of each production were taken to be 0 and the cost of each terminal were taken to be 1, then AO* would find a string containing the minimum number of not necessarily distinct terminals--i.e., a string of minimum length. AO* could probably could be modified to find all such strings rather than just one.

However, it is not clear how to modify AO* to find strings containg the minimum number of distinct terminals. The main problem is whether the cost of solving a subproblem is to be counted once or twice if the subproblem appears twice in a given problem. If we want a string of minimum length, then we want to count the cost twice--and AO* can handle that. If we want a string containing the minimum number of distinct terminals, we want to count the cost once. AO* cannot handle that.

The above difficulty should not be too surprising. In particular, if Problem 2 is a generalization of the set covering problem then Problem 2 is NP-hard. Thus, since AO* operates in polynomial time (probably $O(n^3)$ although we have not determined the exact figure), it would be very surprising if it could solve Problem 2.

Chang and Slagle [3] invented an AND/OR graph search procedure which counts the cost of a subproblem only once if the subproblem appears twice in a given problem. Thus Chang and Slagle's algorithm could be used to find a string containing the minimum possible number of distinct terminals. What Chang and Slagle's procedure would do on the grammar G2 is (in effect) to generate all set covers and then select a minimal one. In general this would take exponential time; however, I think some modifications could be made to the procedure to speed up its average-case performance.

In particular, we think it may be possible to modify it so that it would not generate any set covers larger than it had to. There may be some interesting ways to combine Chang and Slagle's procedure with some of the set covering techniques which one of the principal investigators (Dana Nau) is exploring with James Reggia.

HIGH LEVEL OVERVIEW

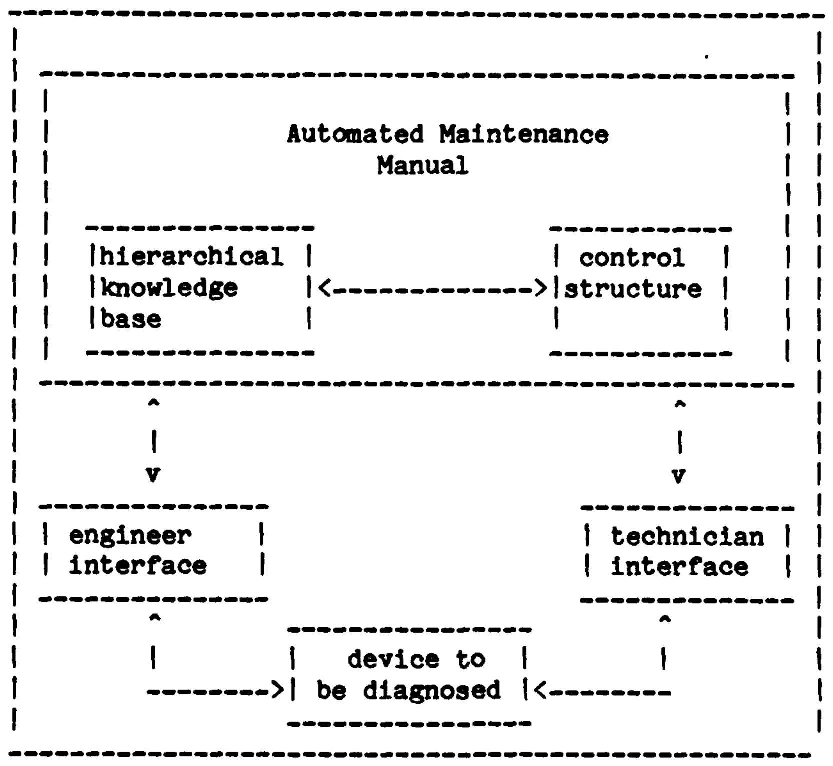


Figure 3.1

ADDER EXAMPLE

test

```
--------------------------------
|  Does register C hold the    |
|  correct sum?                |
|  (yes, no):                  |
--------------------------------
```

text

```
--------------------------------
| An adder is a chip with      |
| three registers A, B, and C. |
| The adder takes the contents |
| of A and B, adds them  to-   |
| gether and transfers the sum |
| to register C.               |
--------------------------------
```

diagram

```
-----------------------------------
|                     ----------- |
|    |----------------|reg  A   | |
|    |   |--------    ----------- |
|    v   v       |    ----------- |
|  ---------  -----|reg  B   |    |
|  | adder |        ----------- |
|  ---------          ----------- |
|   |    |----------->|reg  C   | |
|   v                 ----------- |
|  ---                            |
|  |E|                            |
|  ---                            |
-----------------------------------
```

Figure 3.2

```
              DECISION NODE

                   |
                   |
                   v
------------------------------------------
|                                        |
|                test                    |
|     --------------------------------   |
|     |  Does register C hold the    |   |
|     |  correct sum?                |   |
|     |  (yes, no):                  |   |
|     --------------------------------   |
|                                        |
|                 text                   |
|     --------------------------------   |
|     | An adder is a chip with      |   |
|     | three registers A, B, and C. |   |
|     | The adder takes the contents |   |
|     | of A and B, adds them  to-   |   |
|     | gether and transfers the sum |   |
|     | to register C.               |   |
|     --------------------------------   |
|                                        |
|               diagram                  |
|    ----------------------------------- |
|    |                     ----------- | |
|    |    |----------------|reg  A   | | |
|    |    |   |--------    ----------- | |
|    |    v   v       |    ----------- | |
|    |  ---------     -----|reg  B   | | |
|    |  | adder |          ----------- | |
|    |  ---------          ----------- | |
|    |   |    |----------->|reg  C   | | |
|    |   v                 ----------- | |
|    |  ---                            | |
|    |  |E|                            | |
|    |  ---                            | |
|    ----------------------------------- |
 -----------------------------------------
        |   |    |    |  .   .   . |
        v   v    v    v            v
```

Figure 3.3

```
                       PACKAGE
                       ..........
........................package .......................
.                      .  1.0   .                     .
.                      ..........                     .
.                                                     .
.                      ----------                     .
.                      |node 1.1|                     .
.                      ___________                    .
.                        |    |                       .
.                 ---------- -------------            .
.                 |node 1.2| |node RRN1.1|            .
.                 ---------- -------------            .
.                  |     |                            .
.           ---------- -------------                  .
.           |node 1.3| |node RRN1.2|                  .
.           ---------- -------------                  .
.             |    |                                  .
.     ---------- -------------                        .
.     |node 2.0| |node RRN1.3|                        .
.     ---------- -------------                        .
.......................................................
```

Figure 3.4

AMM SYSTEM EXAMPLE

```
                          ---------
                          |package|
                          |1.0    |
                          ---------
                              |
                          ---------
                          |package|
                          |2.0    |
                          ---------
                              |
                          ---------
                          |package|
                          |3.0    |
                          ---------
                              |
                          ---------
                          |package|
                          |4.0    |
                          ---------
                              |
     ------------------------------------------------------
    |          |          |          |          |          |
---------  ---------  ---------  ---------  ---------  ---------
|package|  |package|  |package|  |package|  |package|  |package|
|  7.0  |  |  8.0  |  | 10.0  |  | 11.0  |  | 12.0  |  | 13.0  |
---------  ---------  ---------  ---------  ---------  ---------
             |      |
       ---------  ---------
       |package|  |package|
       | 9.0   |  |5.0    |
       ---------  ---------
                      |
                  ---------
                  |package|
                  |6.0    |
                  ---------
```

Figure 3.5

AMM DICTIONARY

```
AMM  DICTIONARY
------------------
|package 1.0     |-----------------|
------------------_________________|_____________
|package 2.0     |                 v            v
------------------           PACKAGE 1.0     PACKAGE 2.0 ... PACKAGE 13.0
|package 3.0     |           --------------  -----------     ------------
------------------           |node 1.1    |  |         |     |          |
|package 4.0     |           --------------
------------------           |node 1.2    |  |         |     |          |
|package 5.0     |           --------------
------------------           |node 1.3    |  |---------|     |----------|
|package 6.0     |           --------------                       ^
------------------           |node 2.0    |                       |
|package 7.0     |           --------------                       |
------------------           |node RRN 1.1|                       |
|package 8.0     |           --------------                       |
------------------           |node RRN 1.2|                       |
|package 9.0     |           --------------                       |
------------------           |node RRN 1.3|                       |
|package 10.0    |           --------------                       |
------------------                                                |
|package 11.0    |                                                |
------------------                                                |
|package 12.0    |                                                |
------------------                                                |
|package 13.0    |-------------------------------------------------
------------------
```

Figure 3.6

REFERENCES

[1] Allen, E. M., Yaps: Yet Another Production System, Tech. Report TR-1146, Computer Science Dept., Univ. of Md., College Park, MD, Feb. 1982.

[2] Allen, E. M., Trigg, R. H., and Wood, R. J., The Maryland Artificial Intelligence Group Franz Lisp Environment, Tech. Report TR-1226, Computer Science Dept., Univ. of Md., College Park, MD, Oct. 1982.

[3] Chang, C. L. and Slagle, J. R., An Admissible and Optimal Algorithm for Searching AND/OR Graphs, Artificial Intelligence 2, 2, pp. 117-128, 1971.

[4] Forgy, C. L., The OPS5 User's Manual, Tech. Report, Computer Sci. Dept., Carnegie-Mellon University, 1980.

[5] Kasif, E. H., Automated Maintenance Manual: A Tool for System Diagnosis, M.S. Thesis, Computer Sci. Dept., Univ. of Maryland, College Park, MD, 1983. To appear.

[6] Miller, J., , Personal communication, McDonnell Douglas Corporation, St. Louis, MO, Feb. 1983.

[7] Miller, R., Pople, H., and Myers, J., Internist-1: An Experimental Computer-Based Diagnostic Consultant for General Internal Medicine, NEJM 307, pp. 468-476, 1982.

[8] Nau, D. S., Expert Computer Systems, Computer 16, 2, pp. 63-85, Feb. 1983.

[9] Nau, D. S. and Chang, T. C., Prospects for Process Selection Using Artificial Intelligence, Tech. Report TR-1268, Computer Sci. Dept., Univ. of Md., March 1983. Submitted to Computers in Industry.

[10] Nau, D. S., Reggia, J. A., and Wang, P., Knowledge-Based Problem Solving Without Production Rules, Proc. IEEE 1983 Trends and Applications Conference, pp. 105-108, May 1983.

[11] Nilsson, N. J., Principles of Artificial Intelligence, Tioga, Palo Alto, 1980.

[12] Pople, H. E., The Formation of Composite Hypotheses in Diagnostic Problem Solving: an Exercise in Synthetic Reasoning, Proc. Fifth Internat. Joint Conf. Artif. Intell., pp. 1030-1037, 1977.

[13] Reggia, J. A., Knowledge-Based Decision Support Systems: Development through KMS, Ph.D. Dissertation, Tech. Report TR-1121, Computer Sci. Dept., Univ. of Md., Oct. 1981.

[14] Reggia, J. A., Nau, D. S., and Wang, P., Diagnostic Expert Systems Based on a Set Covering Model, International Journal of Man-Machine

Studies, Oct. 1983. To appear.

[15] Wood, R. J., Franz Flavors: An Implementation of Abstract Data Types in an Applicative Language, Tech. Report TR-1174, Computer Science Dept., Univ. of Md., College Park, MD, June 1982.

FILMED